Blind SQL injection: are your web applications vulnerable?

White paper

Table of contents

Introduction

SQL injection occurs when an application does not properly validate user-supplied input and then includes that input as part of a SQL statement. SQL injection largely depends on an attacker discovering and verifying portions of the original SQL query, using information from error messages. However, web applications can still be vulnerable to blind SQL injection attacks even with no error messages or when they only reveal generic information. By altering the input parameters, an attacker can pose various "true-false" statements to the application to gather information about the database and then ultimately reconstruct the SQL statement by gauging its behavior. Are different pages displayed as a result of changed input? Does an inserted "wait" command cause the application to pause before responding? It is called "blind," because no significant error appears, yet the application is still vulnerable. Blind SQL injection is as dangerous as SQL injection, and it can have the same consequences. This white paper educates security professionals and developers on the techniques they can use to take advantage of a web application that is vulnerable to blind SQL injection, and describes some of the techniques they can use to protect against blind SQL injection and similar input validation problems.

What is blind SQL injection?

In many aspects, SQL injection and blind SQL injection are the same. A common coding error facilitates both blind SQL and SQL injection: The application accepts data from a client and executes SQL queries without first validating the client's input. The attacker is then free to extract, modify, add or delete content from the database. In some circumstances, the attacker may even penetrate past the database server and into the underlying operating system.

A primary difference between the attacks is in the method of determination. Hackers typically test for SQL injection vulnerabilities by sending the application input that causes the server to generate an invalid SQL query. If the server returns an error message to the client, the attacker attempts to reverse-engineer portions of the original SQL query, using information gained from the error messages. A typical administrative safeguard is simply to prohibit the display of database server error messages. The absence of errors only means that the application is protected against one form of SQL injection.

Because blind SQL injection attacks do not rely on error messages, you cannot look for specific patterns or strings in the web server's response. Instead, an attacker looks to see whether two requests with different parameter values return the same information. In essence, blind SQL injection attacks attempt to recreate the query in such a way that the meaning stays the same, but its content differs.

Detecting blind SQL injection

Web applications commonly use SQL queries with client-supplied input in the WHERE clause to retrieve data from a database. By adding additional conditions to the SQL statement and evaluating the web application's output, you can determine whether an application is vulnerable to blind SQL injection.

For example, many companies allow Internet access to archives of their press releases. A URL for accessing the company's fifth press release may appear as:

http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5

The web application may retrieve the press release using the following SQL statement (the client-supplied input is bold):

SELECT title, description, releaseDate, body FROM pressReleases WHERE pressReleaseID = **5**

The database server responds by returning the data for the fifth press release. The web application then formats the press release data into an HTML page and sends the response to the client.

To determine whether an application is vulnerable to blind SQL injection, you can inject an extra true condition into the WHERE clause. For example, you may request this URL:

http://www.thecompany.com/pressRelease.jsp?pressReleaseID=**5 AND 1=1**

The database server executes the following query:

SELECT title, description, releaseDate, body FROM pressReleases WHERE pressReleaseID = 5 AND 1=1

If the query also returns the same press release, then the application is susceptible to blind SQL injection. Part of the user's input is interpreted as SQL code.

A secure application rejects this request because it treats the user's input as a value, and the value 5 AND 1=1 causes a type mismatch error. The server does not display a press release.

Another method of testing for blind SQL injection vulnerabilities is to alter the "math" of the parameter. For example, instead of submitting 5 as the value of PressReleaseID, an attacker may submit 3%2b3, which equals 3 + 2 if the raw string is passed verbatim to the database. The database resolves the query, because it conforms to a valid syntax. If the same press release is returned, the application is vulnerable to blind SQL injection.

You should also make sure that inserting 1=1 does not yield results based on a flaw in the application instead of blind SQL injection. You can do this by inserting 1=2, an untrue condition, into the SQL query. If the results for each query are the same, then SQL injection does not exist.

Exploiting vulnerabilities

When testing for vulnerabilities for blind SQL injection, the injected WHERE condition is predictable: 1=1 is always true. However, when you attempt to exploit a vulnerability, you don't know whether the injected WHERE condition is true or false before sending it. If a record is returned, the injected condition must be true. You can use this behavior to ask the database server true-false questions. For example, the following request asks the database server, "Is the current user dbo?"

http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND USER_NAME() = 'dbo'

USER_NAME() is a SQL Server function that returns the name of the current user. If the current user is dbo (administrator), the fifth press release is returned. If not, the query fails, and no press release displays. By combining subqueries and functions, you can pose more complex questions. The following example tries to retrieve the name of a database table, one character at a time:

http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE xtype='U'), 1, 1))) > 109

The subquery, SELECT, asks for the name of the first user table in the database, which is typically the first thing done in SQL injection exploitation. The substring() function returns the first character of the query's result. The lower() function simply converts the character to lower case. Finally, the ascii() function returns the ASCII value of this character.

If the server returns the fifth press release in response to the URL, you know that the first letter of the query's result comes after the letter "m" (ASCII character 109) in the alphabet. By making multiple requests, you can determine the precise ASCII value.

http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE xtype='U'), 1, 1))) > 116

If no press release is returned, the ASCII value is greater than 109 but not greater than 116. Therefore, the letter is between "n" (110) and "t" (116).

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE xtype='U'), 1, 1))) > 113
```

This is another false statement. You now know that the letter is between 110 and 113.

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE xtype='U'), 1, 1))) > 111
```

The statement is false again. The range is narrowed down to two letters: "n" (110) and "o" (111).

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE xtype='U'), 1, 1))) = 111
```

The server returns the press release, so the statement is true. The first letter of the query's result, and the table's name, is "o." To retrieve the second letter, repeat the process but change the second argument in the substring() function so that the next character of the result is extracted (the change is bold):

```
http://www.thecompany.com/pressRelease.jsp?pressReleaseID=5 AND ascii(lower(substring((SELECT TOP 1 name FROM sysobjects WHERE xtype='U'), 2, 1))) > 109
```

Repeat this process until the entire string is extracted. In this case, the result is "orders."

Solutions

Blind SQL and SQL injection are attacks upon the web application, not the web server or the operating system. Therefore, fixes—other than those implemented in the application code—are stop-gap measures and short-term solutions at best. Most methods for preventing blind SQL and SQL injection also have their own limitations. Therefore, you should employ a layered approach to preventing these attacks and implement several different measures to prevent unauthorized access to your back-end database.

Parameterized queries

SQL injection arises from an attacker's manipulation of query data to modify query logic. Therefore, the best way to prevent both blind SQL and SQL injection attacks is to separate the logic of a query from its data. This prevents commands that are inserted from user input from being executed. The downside of this approach, although slight, is that it can affect performance, and each query on the site must be structured in this method to be completely effective. If one query is inadvertently bypassed, the application can be vulnerable. The following code shows a sample SQL statement that is SQL injectable:

```
sSql = "SELECT LocationName FROM Locations ";
sSql = sSql + " WHERE LocationID = " + Request["LocationID"];
oCmd.CommandText = sSql;
```

Figure 1. Obtaining a product ID and using it in a SQL query

C# sample:

```
string connString =
WebConfigurationManager.ConnectionStrings["myConn"].ConnectionString;
using (SqlConnection conn = new SqlConnection(connString))
{
conn.Open();
SqlCommand cmd = new SqlCommand("SELECT Count(*) FROM Products
WHERE ProdID=@pid", conn);
SqlParameter prm = new SqlParameter("@pid", SqlDbType.VarChar,50);
prm.Value = Request.QueryString["pid"];
cmd.Parameters.Add(prm);
int recCount = (int)cmd.ExecuteScalar();
}
```

VB.NET sample:

```
Dim connString As String =
WebConfigurationManager.ConnectionStrings("myConn").ConnectionString
Using conn As New SqlConnection(connString)
conn.Open()
Dim cmd As SqlCommand = New SqlCommand("SELECT Count(*)FROM
Products WHERE ProdID=@pid", conn)
Dim prm As SqlParameter = New SqlParameter("@pid", SqlDbType.VarChar,
50)
prm.Value = Request.QueryString("pid")
cmd.Parameters.Add(prm)
Dim recCount As Integer = cmd.ExecuteScalar()
End Using
```

The following example uses parameterized queries and is safe from SQL injection attacks:

```
sSql = "SELECT * FROM Locations ";
sSql = sSql + " WHERE LocationID = @LocationID";
oCmd.CommandText = sSql;
oCmd.Parameters.Add("@LocationID",
Request["LocationID"]);
```

The application sends the SQL statement to the server without including the user's input. Instead, a parameter, @LocationID, is used as a placeholder for the input. In this way, user input never becomes part of the command that SQL executes. Any input that an attacker inserts is effectively negated. An error is still generated, but it is a simple data-type conversion error and not something that an attacker can exploit.

The code examples in Figure 1 illustrate a product ID being obtained from an HTTP query string and used in a SQL query. The string containing the SELECT statement and passed to SqlCommand is a static string, and it is not concatenated from input. Additionally, the input parameter is passed using a SqlParameter object, whose name, @pid, matches the name used within the SQL query.

Stored procedures

Another method of separating query logic from its data is by using stored procedures to isolate the web application from SQL. To secure an application against blind SQL injection, you must prevent client-supplied data from modifying the syntax of SQL statements. All SQL statements required by the application can be sequestered in stored procedures and kept on the database server. Simply moving all SQL statements into stored procedures does not solve blind SQL and SQL injection problems if you use input parameters without first validating the data.

Stored procedures let you build dynamic SQL statements using string concatenation that can then be executed using EXEC commands. However, this defeats using stored procedures for security purposes if you use input parameters without sanitizing the data first. If you must use arbitrary statements, you can use PreparedStatements. Using PreparedStatements and stored procedures to compile the SQL statement before user input is added makes it impossible for user input to modify the SQL statement. Finally, the application should execute the stored procedures using a safe interface, such as JDBC CallableStatement or ADO Command Object.

Using pressRelease.jsp as an example, the code may look like:

```
String query = "SELECT title, description, releaseDate,
body
FROM pressReleases WHERE pressReleaseID = " +
request.getParameter("pressReleaseID");
Statement stmt = dbConnection.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

The first step toward securing the code is to remove the SQL statement from the web application and put it in a stored procedure on the database server:

```
CREATE PROCEDURE getPressRelease
@pressReleaseID integer
AS
SELECT title, description, releaseDate, body FROM
pressReleases
WHERE pressReleaseID = @pressReleaseID
```

At the application, instead of string building a SQL statement to call the stored procedure, you can use a CallableStatement to safely execute it:

```
CallableStatement cs =
dbConnection.prepareCall("{callgetPressRelease(?)}");
cs.setInt(1,Integer.parseInt(request.getParameter("pressRele
aseID")));
ResultSet rs = cs.executeQuery();
```

In a .NET application, the change is similar. The ASP.NET code is vulnerable to blind SQL injection:

```
String query = "SELECT title, description, releaseDate,
body FROM pressReleases WHERE pressReleaseID = "
+Request["pressReleaseID"];
SqlCommand command = new
SqlCommand(query,connection);
command.CommandType = CommandType.Text;
SqlDataReader dataReader = command.ExecuteReader();
```

As with JSP code, the SQL statement must be converted to a stored procedure, which can then be accessed safely by a stored procedure SqlCommand:

```
SqlCommand command = new
SqlCommand("getPressRelease",connection);
command.CommandType =
CommandType.StoredProcedure;
command.Parameters.Add("@PressReleaseID",SqlDbType.I
nt);
command.Parameters[0].Value =
Convert.ToInt32(Request["pressReleaseID"]);
SqlDataReader dataReader = command.ExecuteReader();
```

Data sanitization

You can prevent the majority of blind SQL injection vulnerabilities by properly validating user input for both type and format. Cleanse all client-supplied data of any characters or strings that can be used maliciously, and do so for all applications, not just those that use SQL queries. The best method to use is "white listing." Only accept certain data for specific fields, such as limiting user input to account numbers or account types for those relevant fields or only accepting integers or letters of the English alphabet for others. Many developers try to validate input by "black listing" characters or "escaping" them. Black listing rejects known bad data, such as a single quotation mark, by placing an "escape" character in front of it so that the following items are treated as a literal value. Stripping quotes or putting backslashes in front of them is not enough and is not as effective as white listing, because it is impossible to know all forms of bad data in advance.

A good method of filtering data is to use a default-deny regular expression. You should only include the type of characters that you want. For example, the following regular expression returns only letters and numbers:

```
s/[^0-9a-zA-Z]//\
```

Make your filter narrow and specific. Whenever possible, use only numbers, and after that, use only numbers and letters. If you need to include symbols or punctuation, make absolutely sure that you convert them into HTML substitutes, such as &quote; or >. For example, if a user submits an e-mail address, allow only the "at" sign, underscore, period and hyphen in addition to numbers and letters and only after those characters are converted to their HTML substitutes.

Database considerations

Limit the rights of database users. Successful blind SQL injection attacks run in the context of the users' credentials. While limiting privileges does not prevent SQL injection attacks outright, it can make them significantly more difficult. Don't give users access to all system-stored procedures if they only need access to a few, user-defined procedures.

You should also have a strong system administrator (SA) password policy. Often, an attacker needs the functionality of the administrator account to utilize specific SQL commands. It is much easier to "brute force" a weak SA password, and it increases the likelihood of a successful blind SQL injection attack. Never use the SA account as the application database user. Instead, create specific accounts for individual purposes. Also, if you do not need them, delete SQL-stored procedures, such as master.Xp_cmdshell, xp_startmail, xp_sendmail and sp_makewebtask.

To learn more, visit www.hp.com/go/software

4AA1-5382ENW, October 2007